TDDE68 + TDDE47

Lecture 3:
Processes, Threads and File Systems (I)

Klas Arvidsson

Based on slides by Mikael Asplund and Adrian Pop

Thanks to Christoph Kessler and Simin Nadjm-Tehrani for some of the material behind these slides.

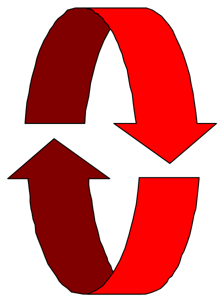
Today's lecture

- Processes
  - Concepts
  - Creation, switching and termination
- Threads
- Process interaction
  - 2 slide introduction
- File Systems (I)
  - Introduction

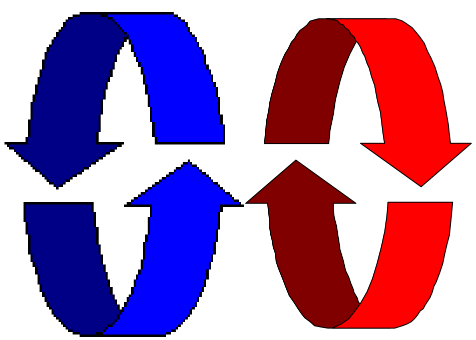
Reading guidelines

- Silberschatz et al. (10th ed.)
  - Chapter 3.1-3.4, 4.1-4.3, 4.5, 13.1, 14.1-2

Concurrent Programs



A **sequential** program has a single thread of control.



A **concurrent** program has multiple threads of control allowing it perform multiple computations "in parallel" and to control multiple external activities which occur at the same time.

[Magee and Kramer 2006]

Related terms

• Concurrent programs	• Define actions that *may* be performed simultaneously
• Parallel programs	• A concurrent program that is designed for execution on parallel hardware
• Distributed programs	• Parallel programs designed to run on network of autonomous processors that do not share memory

Concurrency on a single core CPU

- Concurrency can be achieved through *preemptive multitasking*
- Let each program run for a short while and then switch to the next one
- Improvement over the ”multiprogramming” paradigm

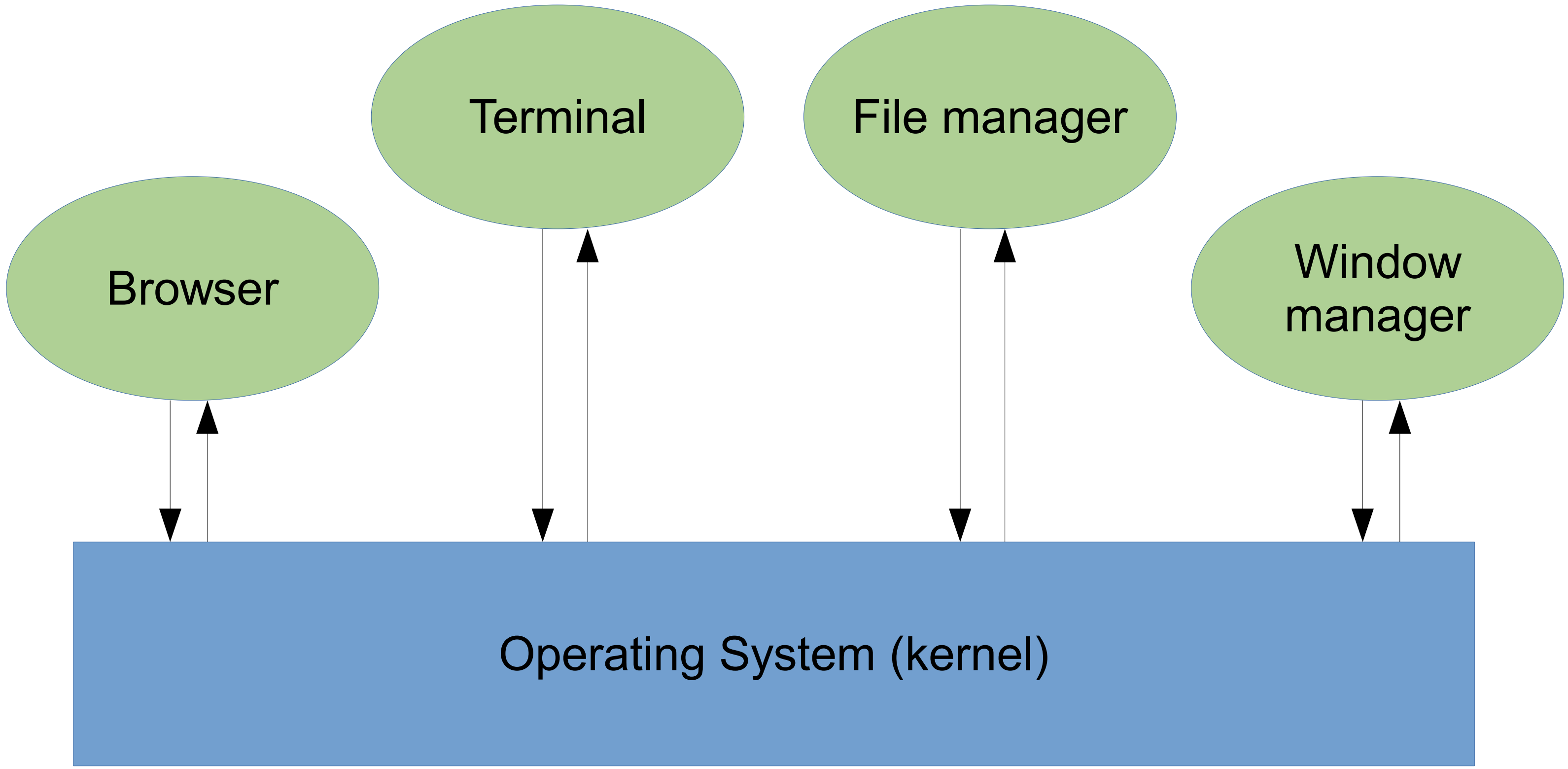
The abstract notion of **Process**

- An abstraction in computer science used for describing program execution and potential parallelism
- What other abstractions do you know?
  - Functions
  - Classes, Objects, Methods
- Processes emphasise *the run-time behaviour*

Typical OS terminology:

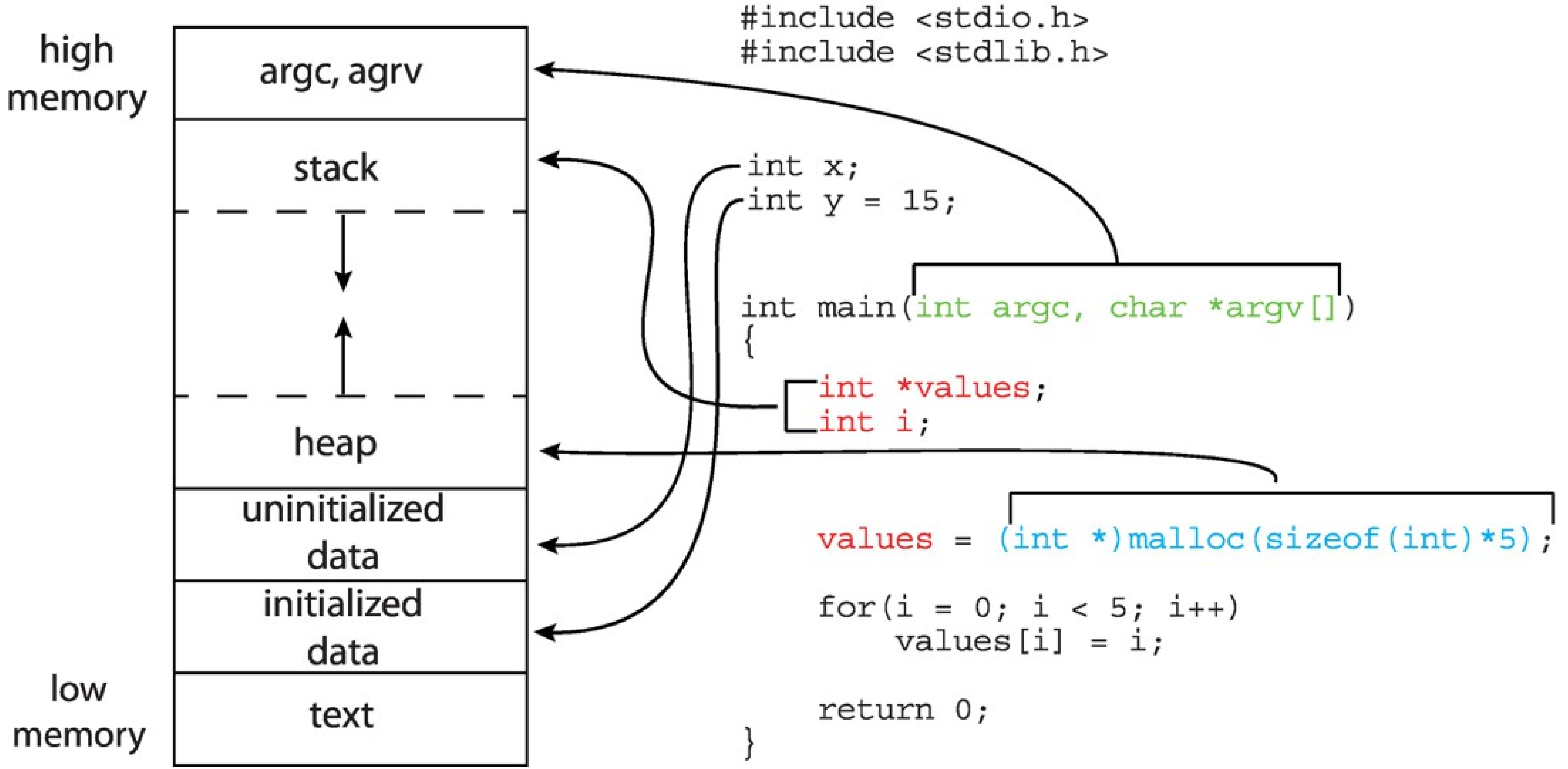
A process is a program in execution with its own memory

Example processes



Process management is one of the key tasks of an operating system.

Process in Memory



Process Control Block (PCB)

Information associated with each process

- Process Identifier (PID)
- Process state
- CPU registers
- Program counter
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process representation in Linux

https://github.com/torvalds/linux/blob/master/include/linux/sched.h

```c
/* Used in tsk->state: */
#define TASK_RUNNING                    0x0000
#define TASK_INTERRUPTIBLE              0x0001
#define TASK_UNINTERRUPTIBLE            0x0002
#define __TASK_STOPPED                  0x0004
#define __TASK_TRACED                   0x0008
/* Used in tsk->exit_state: */
#define EXIT_DEAD                       0x0010
#define EXIT_ZOMBIE                     0x0020
#define EXIT_TRACE                      (EXIT_ZOMBIE | EXIT_DEAD)
/* Used in tsk->state again: */
#define TASK_PARKED                     0x0040
#define TASK_DEAD                       0x0080
#define TASK_WAKEKILL                   0x0100
#define TASK_WAKING                     0x0200
#define TASK_NOLOAD                     0x0400
#define TASK_NEW                        0x0800
/* RT specific auxilliary flag to mark RT lock waiters */
#define TASK_RTLOCK_WAIT                0x1000
#define TASK_STATE_MAX                  0x2000

/* Convenience macros for the sake of set_current_state: */
#define TASK_KILLABLE                   (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED                    (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED                     (TASK_WAKEKILL | __TASK_TRACED)

#define TASK_IDLE                       (TASK_UNINTERRUPTIBLE | TASK_NOLOAD)

/* Convenience macros for the sake of wake_up(): */
#define TASK_NORMAL                     (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)

/* get_task_state(): */
#define TASK_REPORT                     (TASK_RUNNING | TASK_INTERRUPTIBLE | \
                                         TASK_UNINTERRUPTIBLE | __TASK_STOPPED | \
                                         __TASK_TRACED | EXIT_DEAD | EXIT_ZOMBIE | \
                                         TASK_PARKED)

#define task_is_running(task)           (READ_ONCE((task)->__state) == TASK_RUNNING)
```

Diagram of Process State

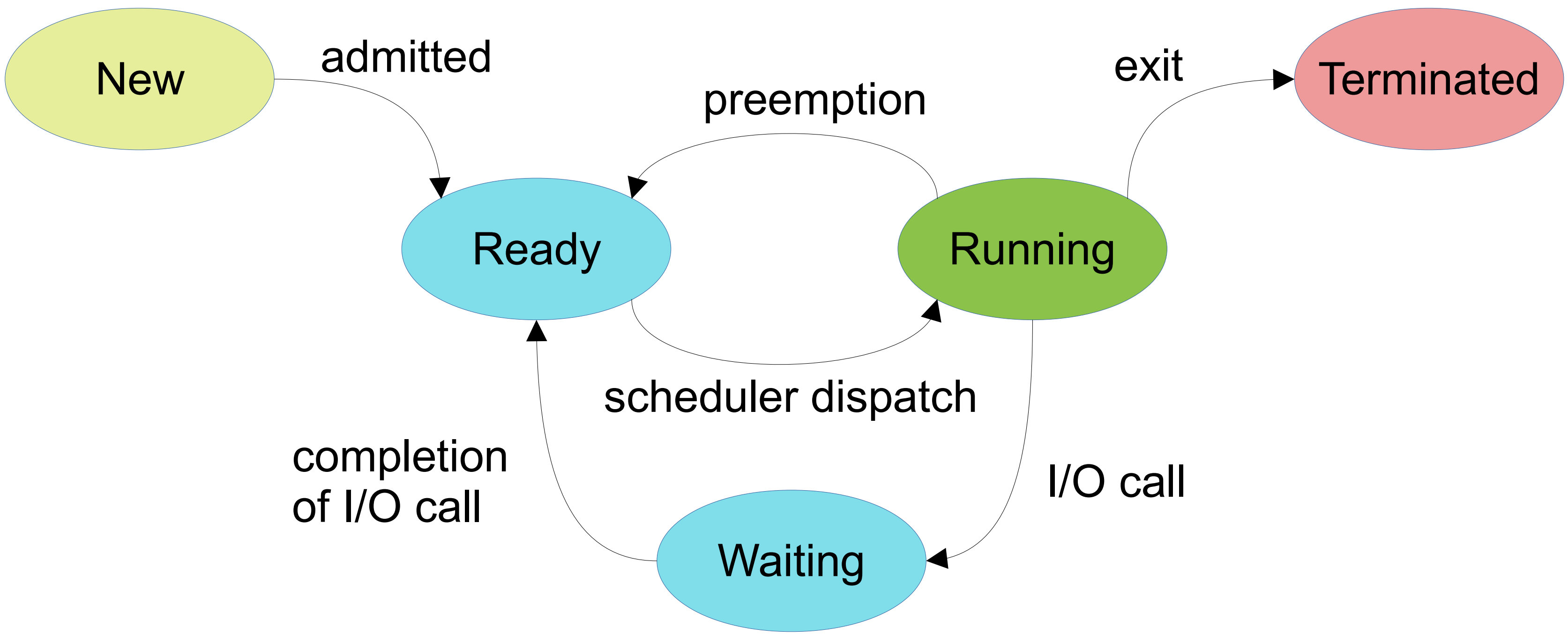
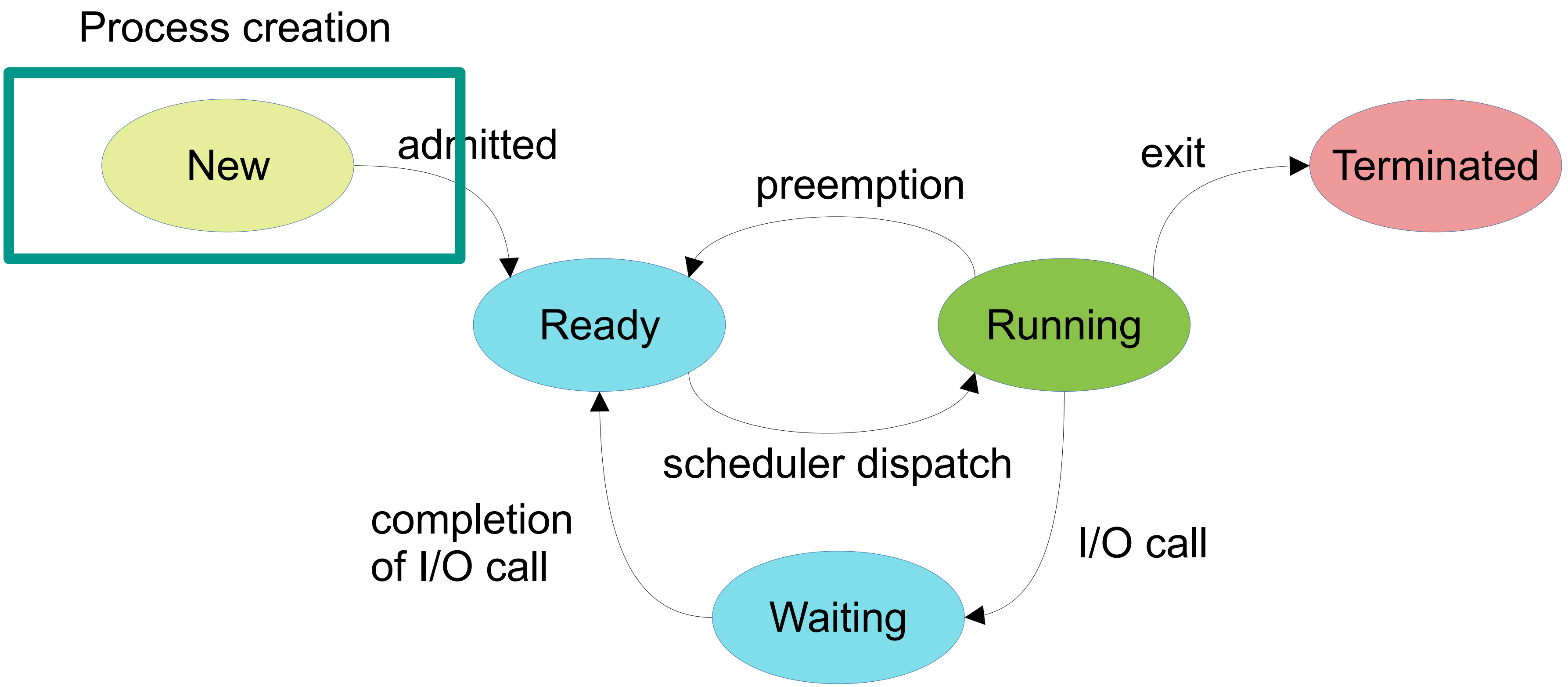
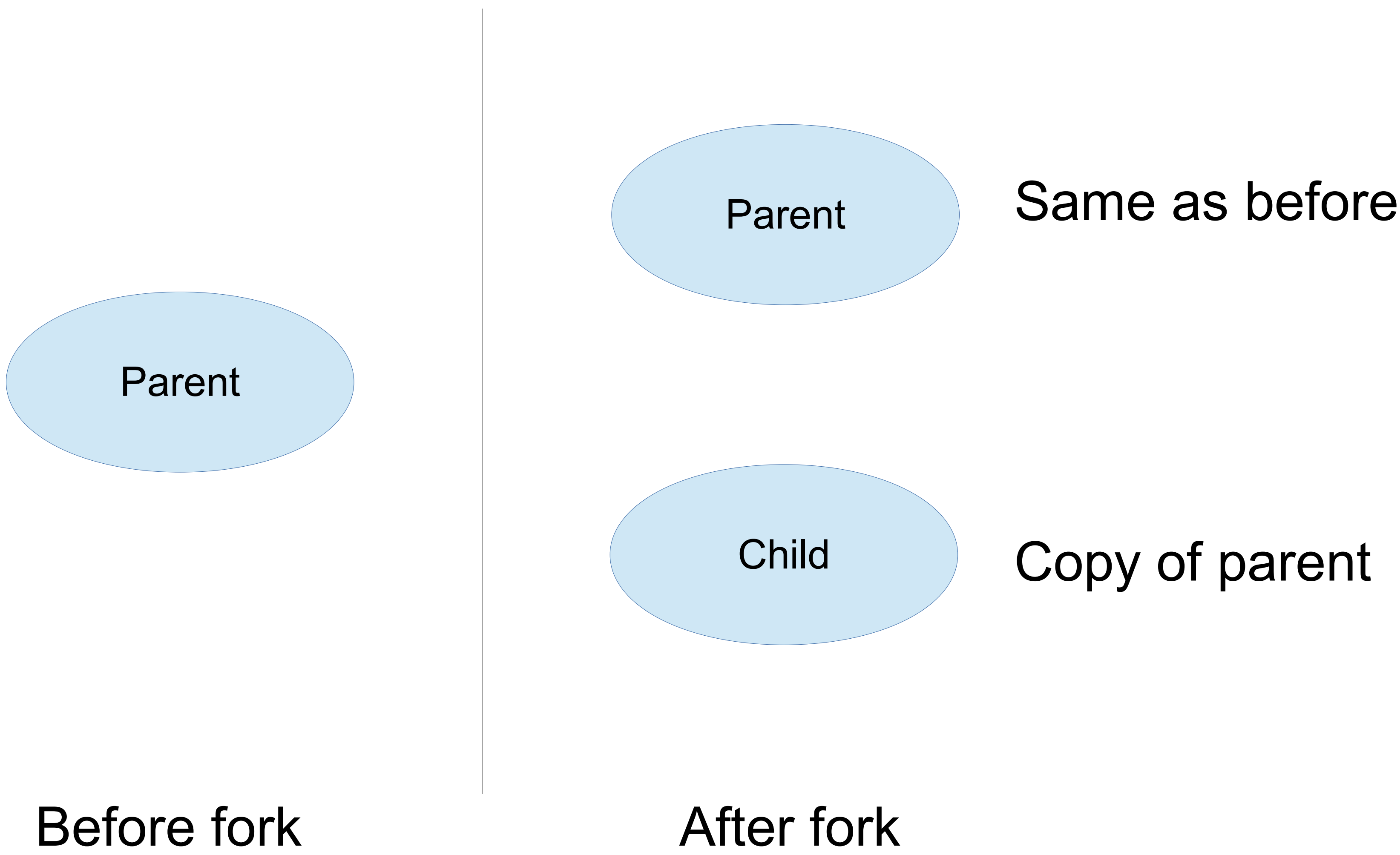


Diagram of Process State

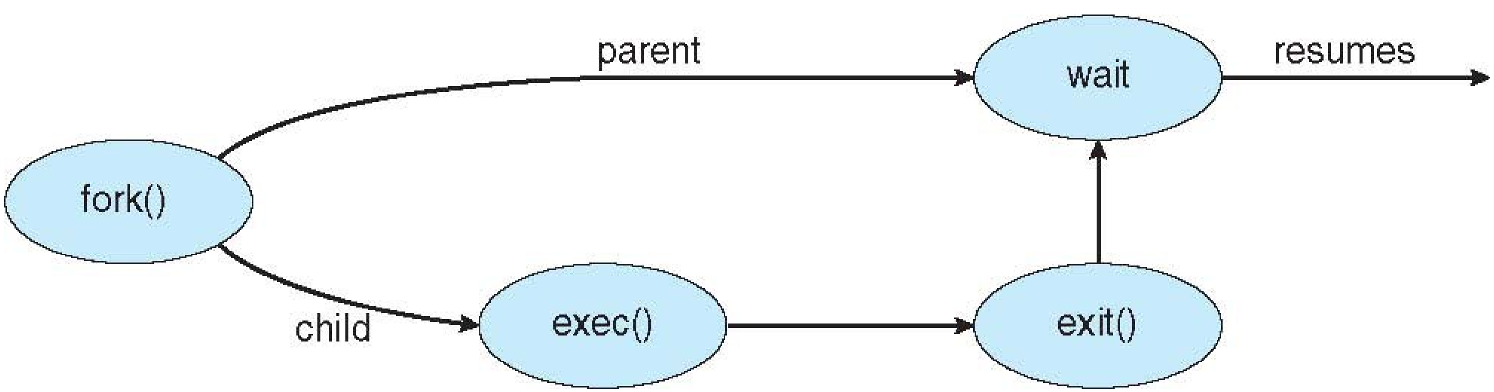


Process Creation



Unix example

- *fork*() system call creates new process
- *exec*() system call used after a fork() to replace the process’ memory space with a new program



More on fork

- fork returns the process id of the child process
  - This is the only way for a process to know if it is the parent or child after a fork

    int pid = 0;

    pid = fork();

    // **for child:** pid == 0, **for parent:** pid == <child process id>

- In Pintos fork is integrated in exec (not in normal Unix)

Poll question

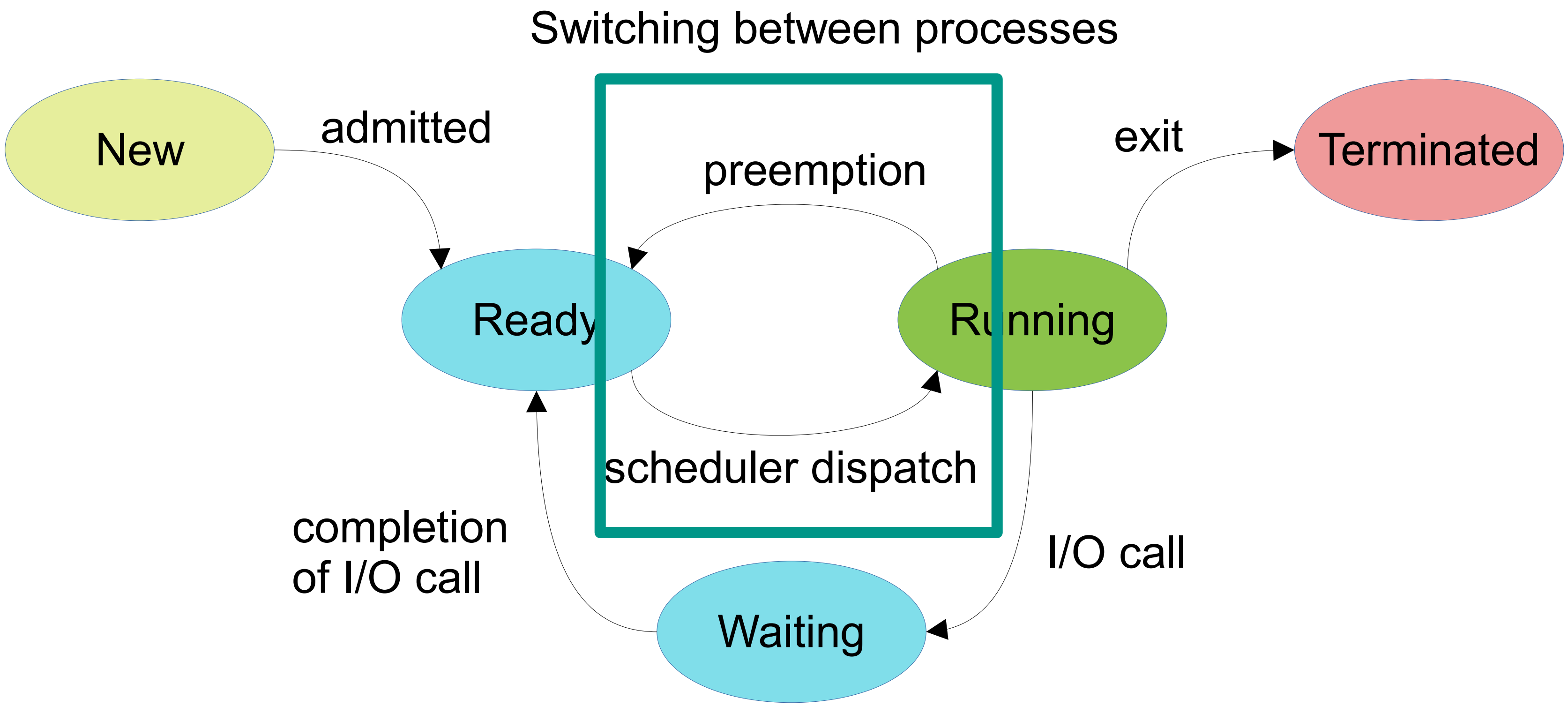
```c
int main(void) {
  int pid1 = 0;
  pid1 = fork();
  if (pid1 == 0) {
    printf("Hello\n");
  }
  int pid2 = fork();
  printf("Hello\n");
  sleep(1);
  return 0;
}
```

How many times will the program output the line ”Hello”?

URL: https://www.menti.com

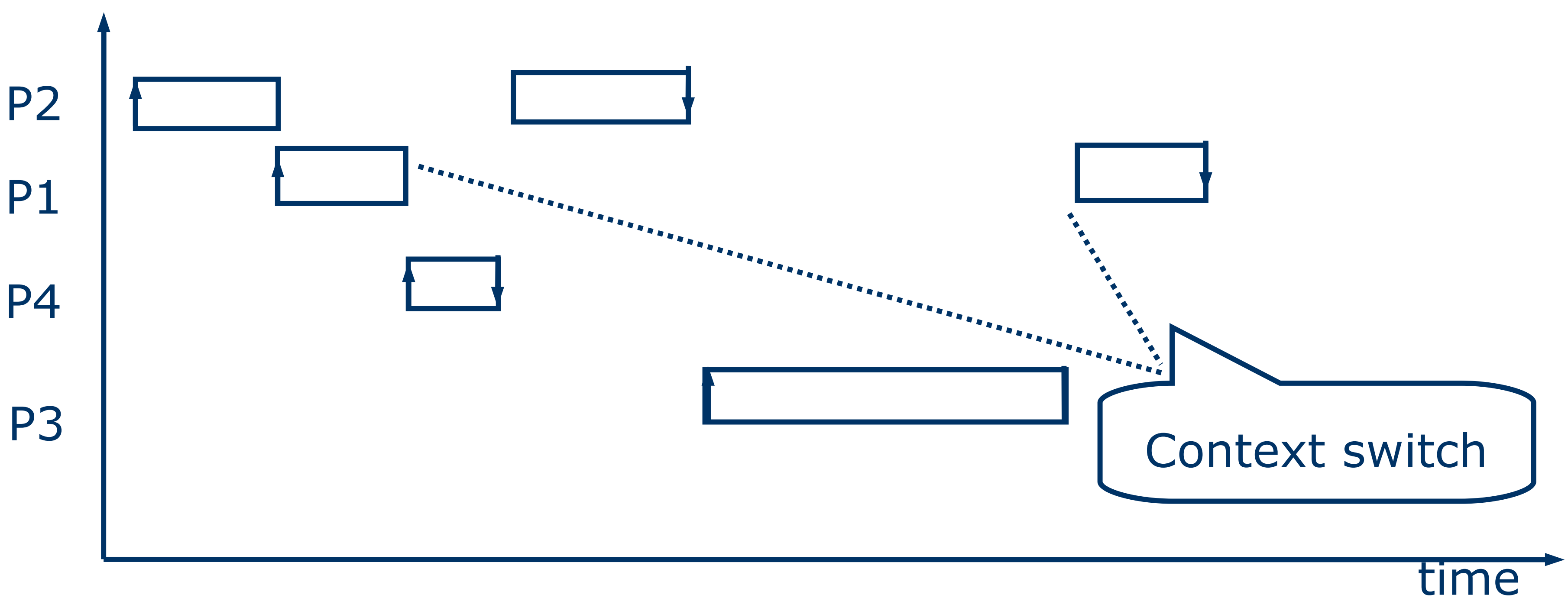
Code: 8619 0811

Diagram of Process State



Context Switch

- Consider a program has several processes P1, …, P4
- An execution of the concurrent program may look like:



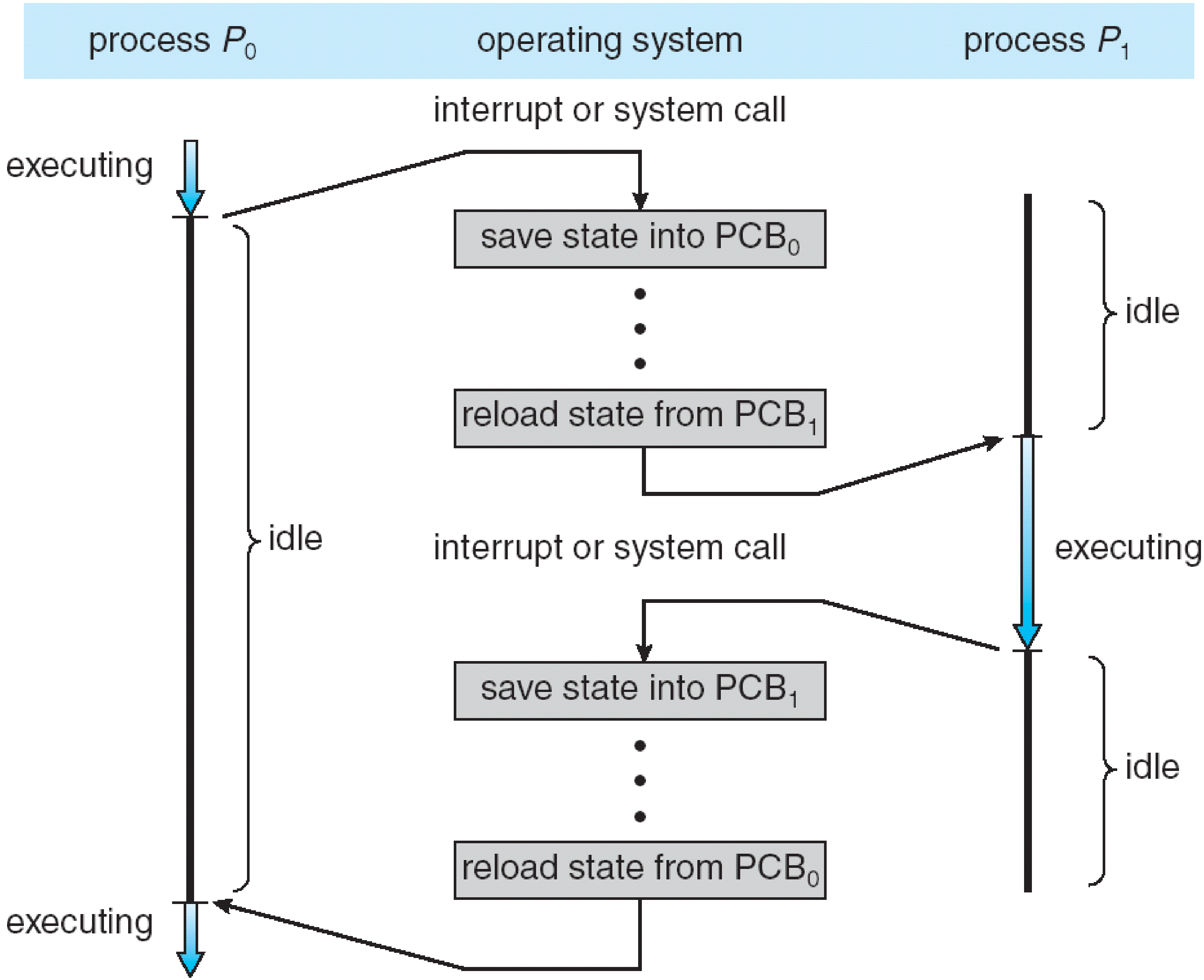
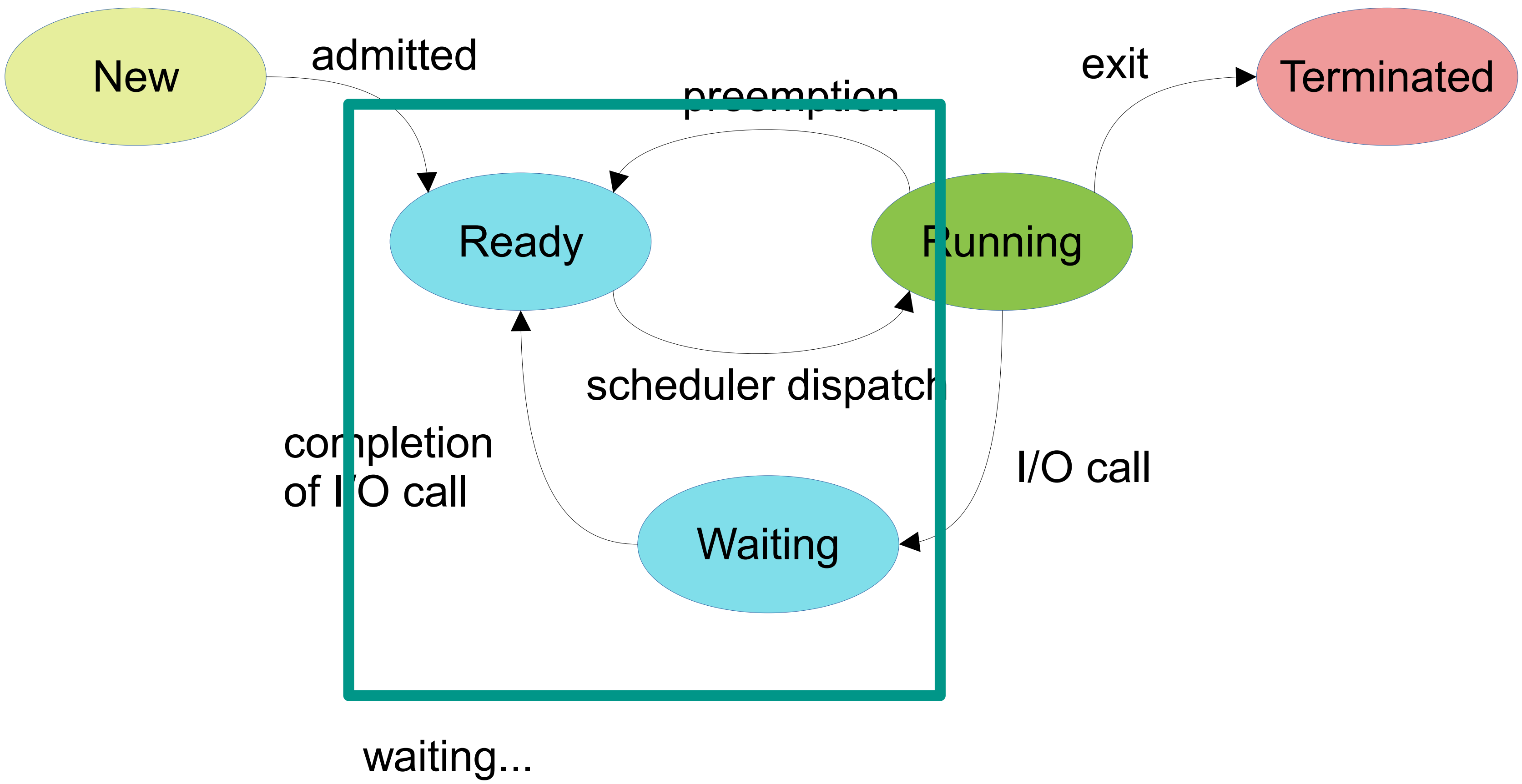
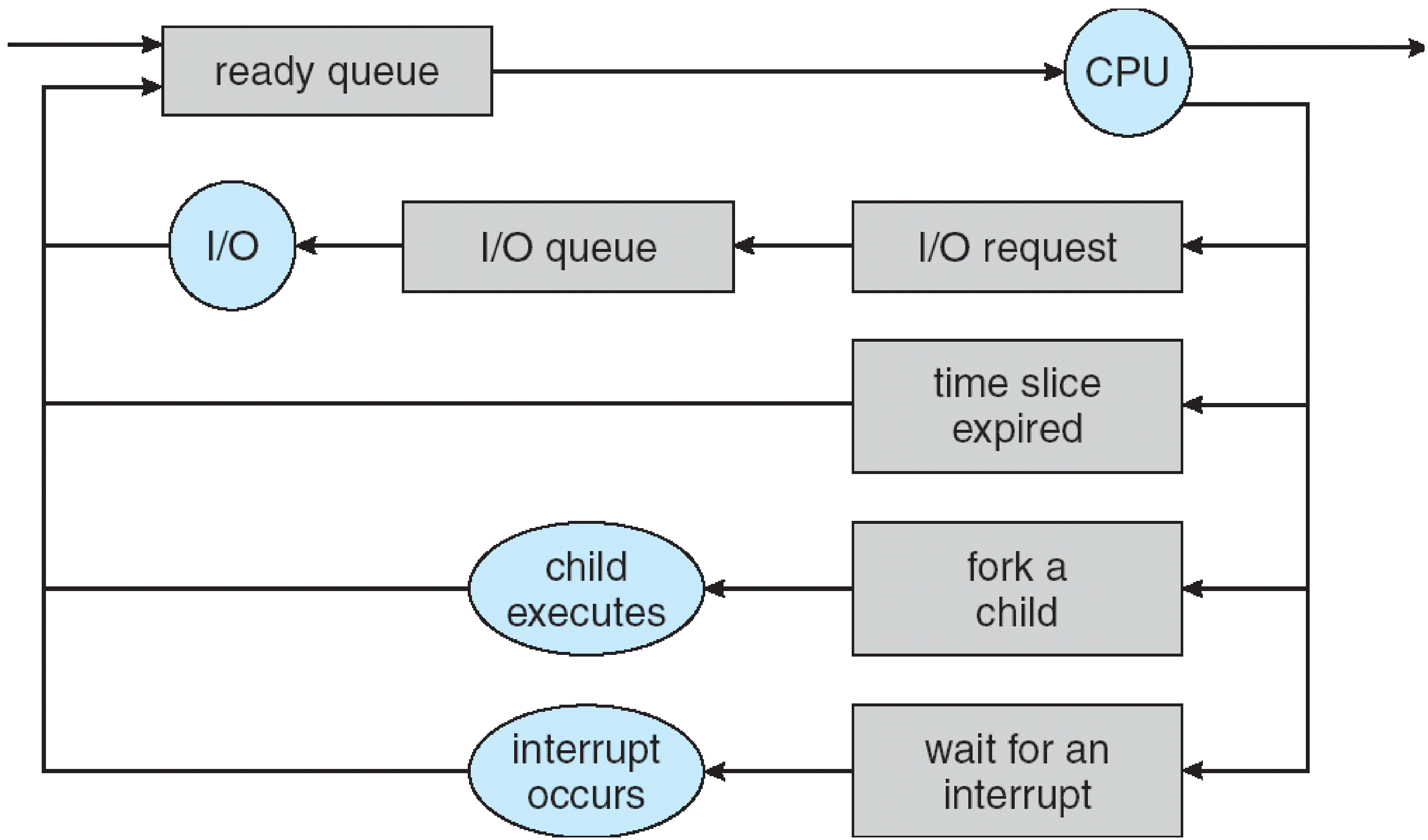

process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
idle
idle
reload state from PCB_1
interrupt or system call
executing
save state into PCB_1
idle
reload state from PCB_0
executing

Diagram of Process State



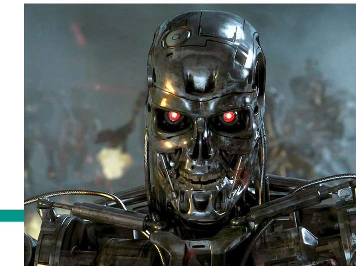
What are processes typically doing?

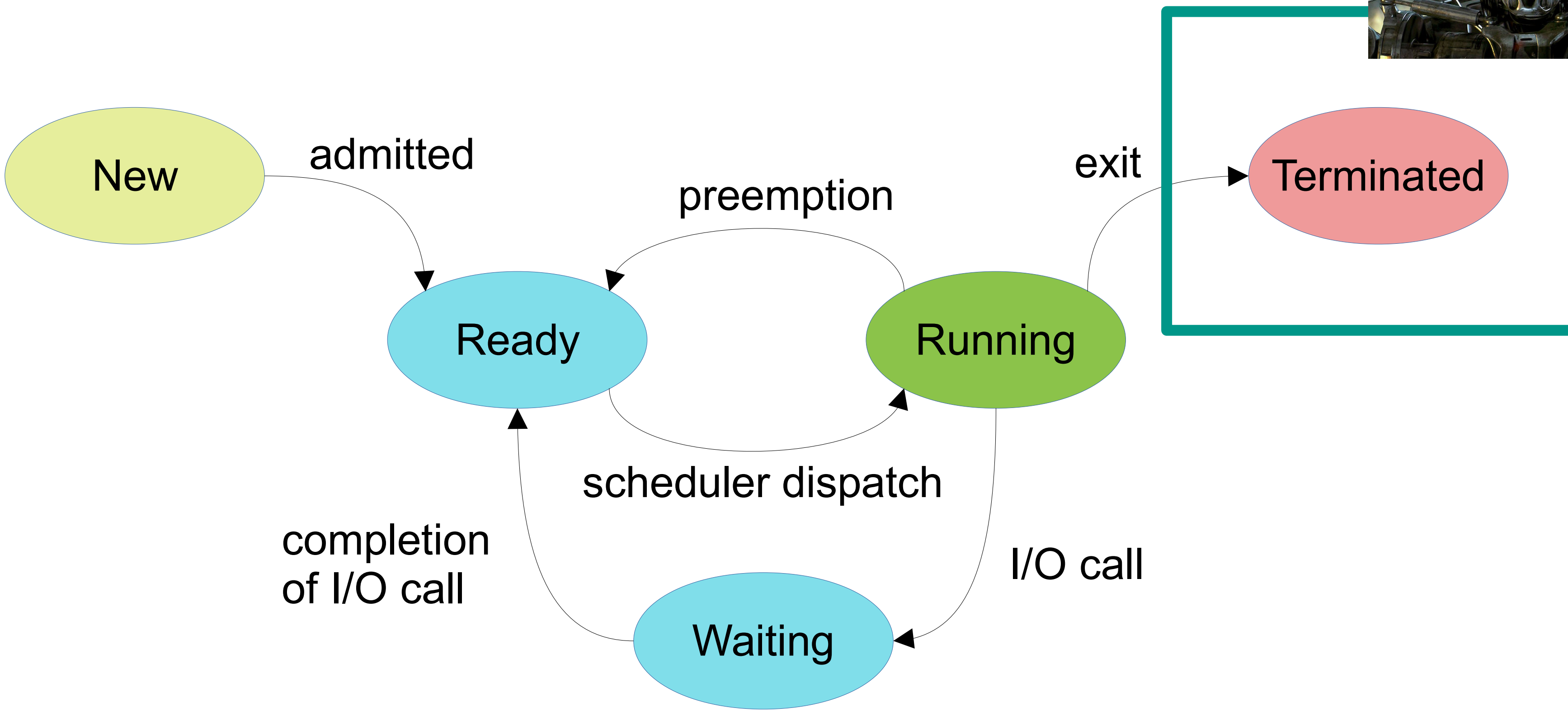
Process queues



For every reason to wait there is a queue.
And then there is the ready queue.

Diagram of Process State





Process termination

- Exit a process by calling exit(EXIT_SUCCESS)
  - (same as return 0)

- OS able to free resources
  - Take back memory, close open files, etc
  - All of it? No!

Parents must care for their children

- Using processes to run a task in the background:

```
pid_t pid = fork();
if (pid == 0) { // This is run by the child
    // Do something useful
} else { // This is run by the parent
    // Continue with own stuff
    int status;
    wait(&status) // Wait until child is done
}
```

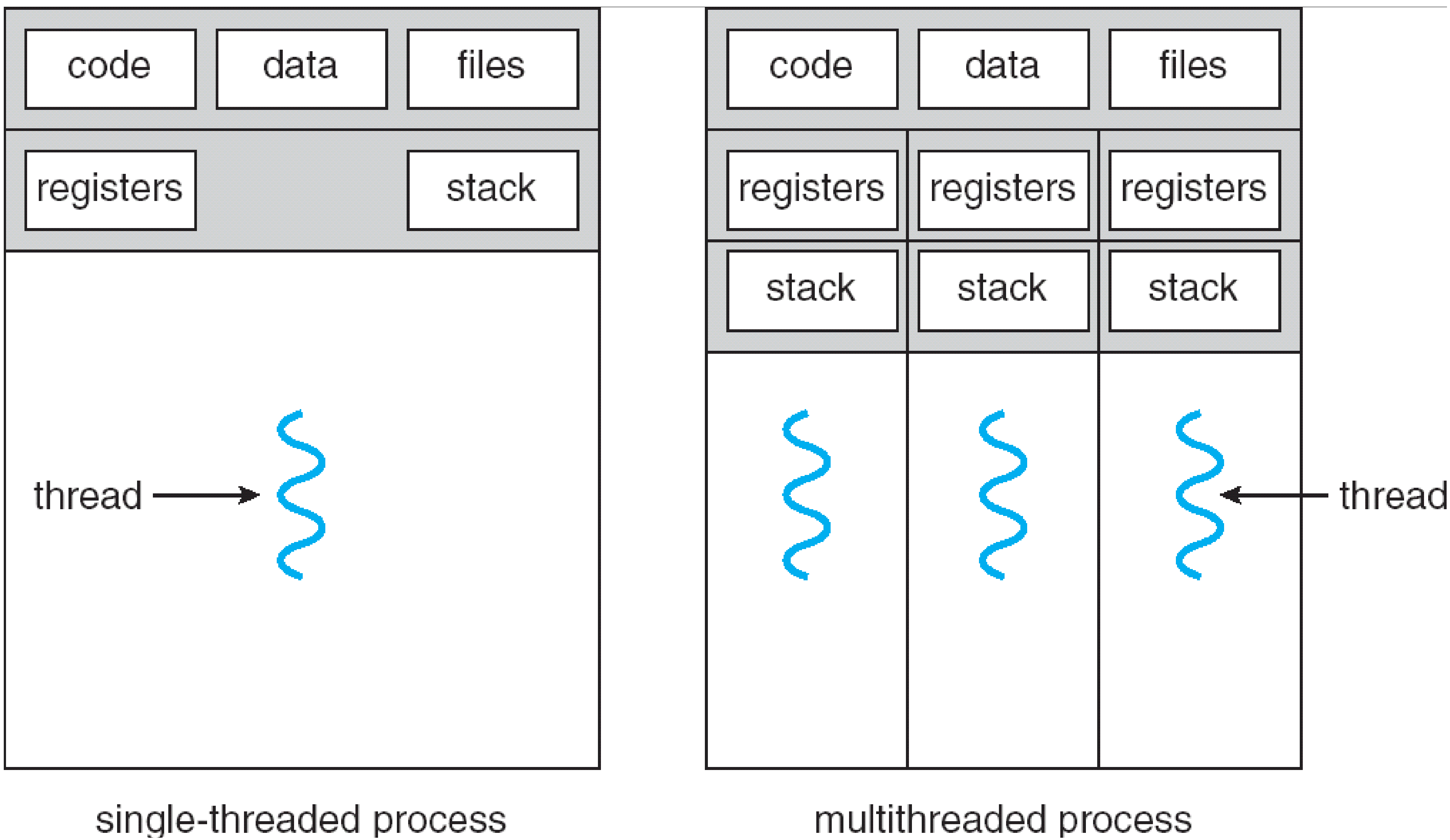
What happens if

- The parent does not call wait?
  - The finished child becomes a **zombie** process

- The parent terminates before the child?
  - The child becomes an **orphan** process (can be adopted by the init process)

Threads

Single- and Multithreaded Processes



Benefits of Multithreading

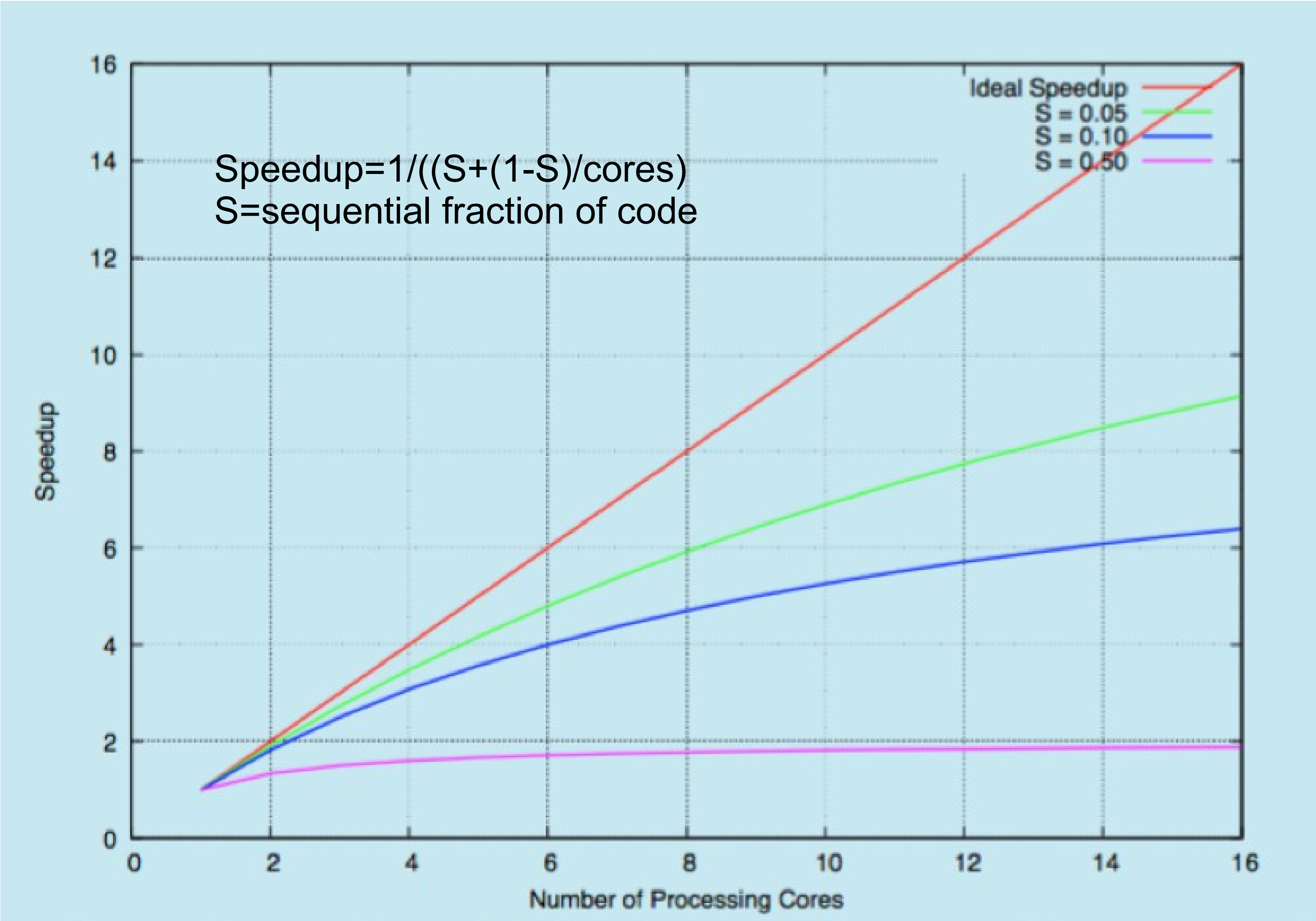
- Responsiveness
  - Interactive application can continue even when part of it is blocked
- Resource Sharing
  - Threads of a process share its memory by default.
- Economy
  - Light-weight
  - Creation, management, context switching for threads is much faster than for processes
    - E.g. Solaris: creation 30x, switching 5x faster
- Utilization of multicore architectures

Multi-core architectures

- Necessitated by the end of Moore’s Law
  - We can no longer keep making chips smaller (and thereby faster)

- Problem: Amdahl’s Law...

Multi-core architectures

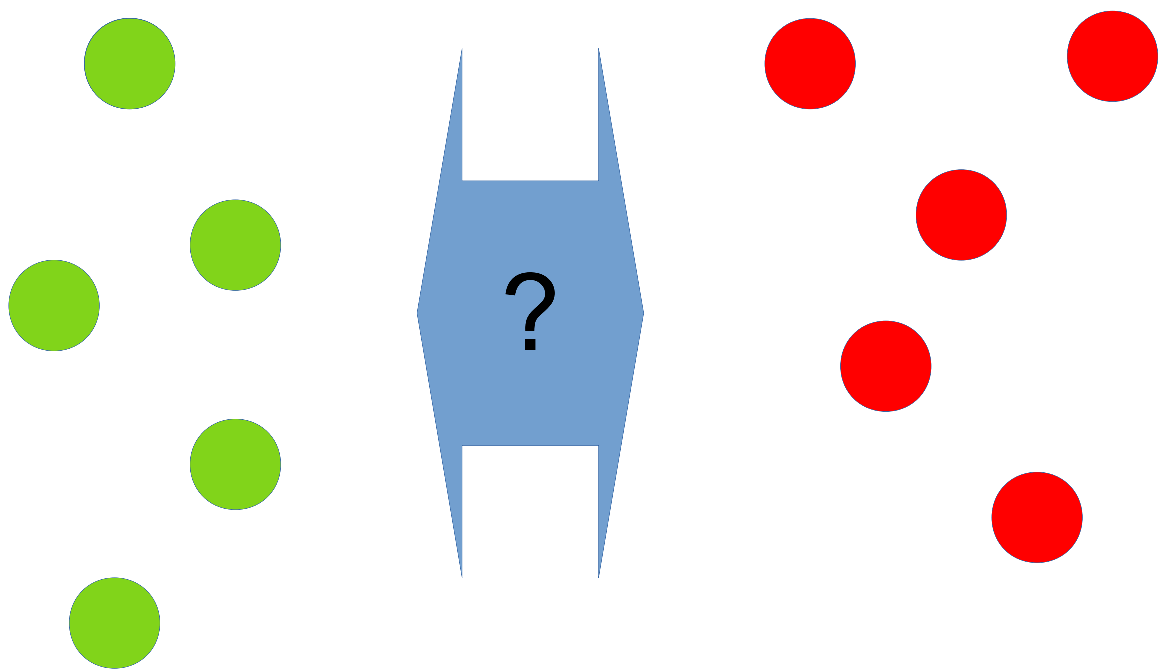


Different perspectives on threads

User space

Kernel space

User-level threads: pieces of code that appear to be running concurrently



Kernel-level threads: entities managed by the scheduler in the kernel

Can be provided by
Runtime environment of programming language
Special libraries
Operating system

Kernel-level threads

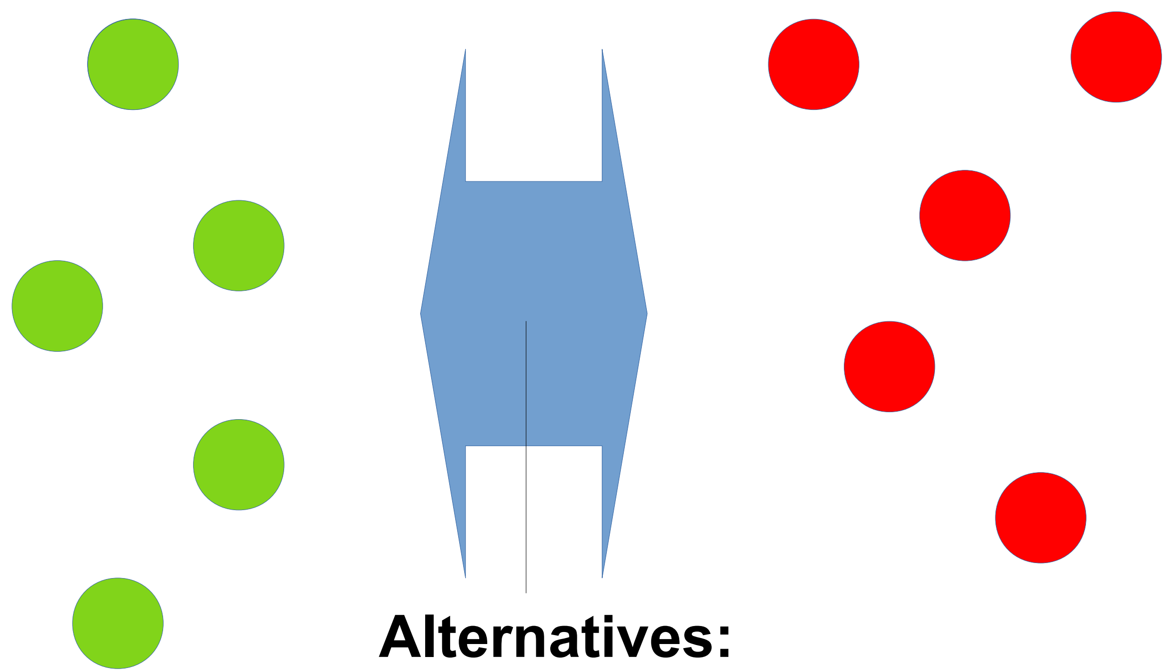
≠

Kernel threads

Mapping from user to kernel

User-level threads

Kernel-level threads



Alternatives:

Many-to-One (M:1)

One-to-One (1:1)

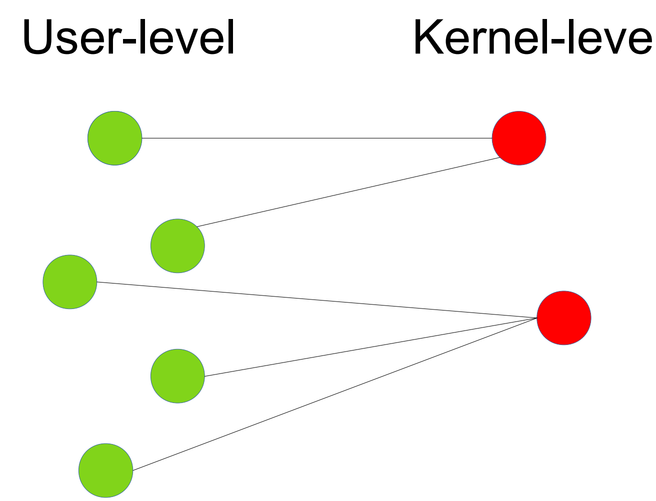
Many-to-Many (M:N)

Many-to-One

- Many user-level threads mapped to single kernel thread

☺ Low overhead, very portable

☹ Not scalable to multiprocessors, does not handle blocking calls well



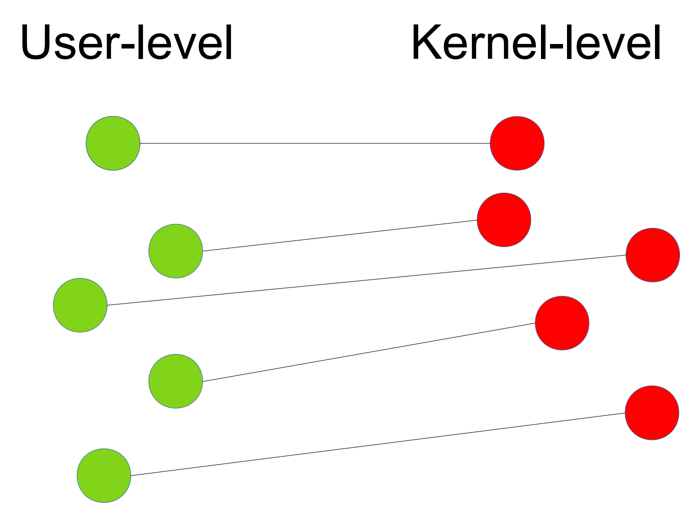
- Example:
  - GNU Portable Threads

One-to-One

- Each user-level thread maps to one kernel thread

☺ more concurrency; scalable to multiprocessors

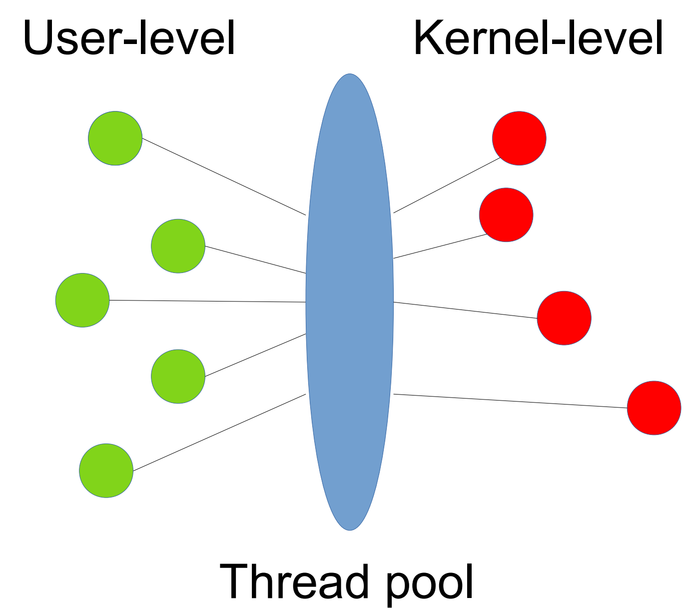
☹ overhead of creating a kernel thread for each user thread



- Examples
  - Windows
  - Unix

Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the OS to create a sufficient number of kernel threads
- Abandoned by most OS:s
  - But used by threading libraries
  - Android, Java, ..



Java example

Implementing Runnable interface:

```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

Creating a thread:

```
Thread worker = new Thread(new Task());
worker.start();
```

Waiting on a thread:

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

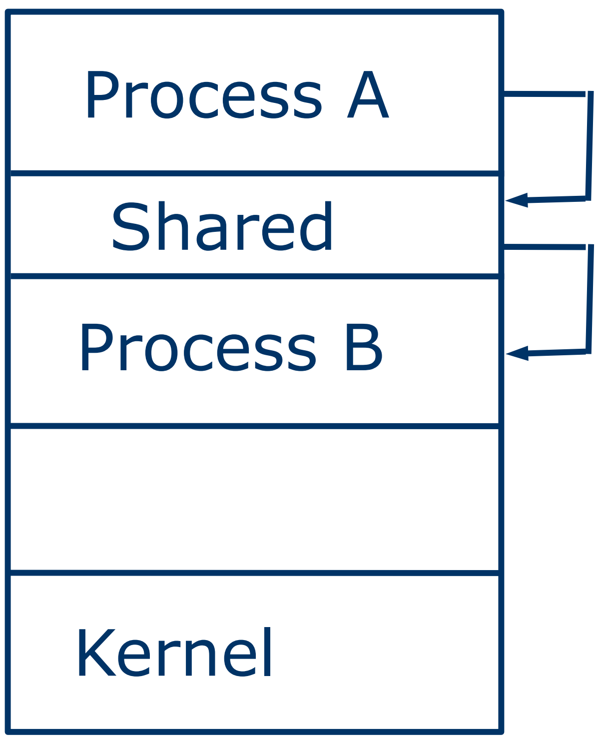
Implicit threading

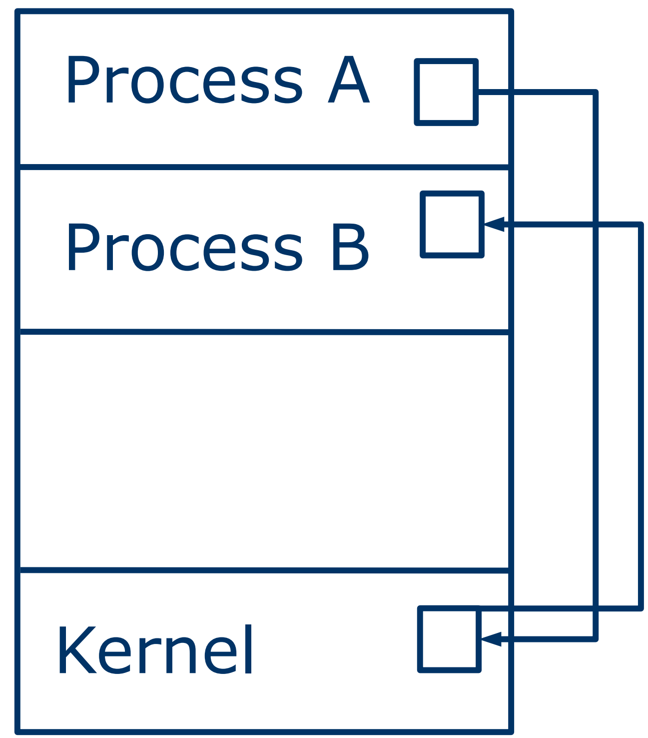
- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Examples
  - Thread Pools (Android, Grand Central Dispatch)
  - Fork-Join (cf. MapReduce)
  - OpenMP
  - Execution policies (C++)

Process interaction

Inter-process communication (IPC)

- Two modes:
  - Shared variables
  - Message passing





Message passing model

- Resembles a distributed system
- Benefits
  - Clean separation of data
  - Easy to distribute across multiple computers
  - Low risk of data corruption
- Drawbacks
  - Distributed algorithms are difficult to construct
  - Might result in bad performance

File systems (I)

File system consists of
interface + implementation

Storing data

- Primary memory is volatile
  - need secondary storage for long-term storage
- A *disk* is essentially a linear sequence of numbered blocks
  - With 2 operations: write block *b*, read block *b*
  - Low level of abstraction

The **file** abstraction

- Provided by the OS
- Smallest allotment of secondary storage known to the user
  - Typically, contiguous logical address space
- Organized in a *directory* of files
- Has
  - Attributes (Name, id, size, …)
  - API (operations on files and directories)

Meta data

- File attributes – name, date of creation, protection info, ...

- Such information *about* files (i.e., **meta-data**) is kept in a **directory structure**, which is maintained on the disk.

- Stored in a **File Control Block** (**FCB**) data structure for each file

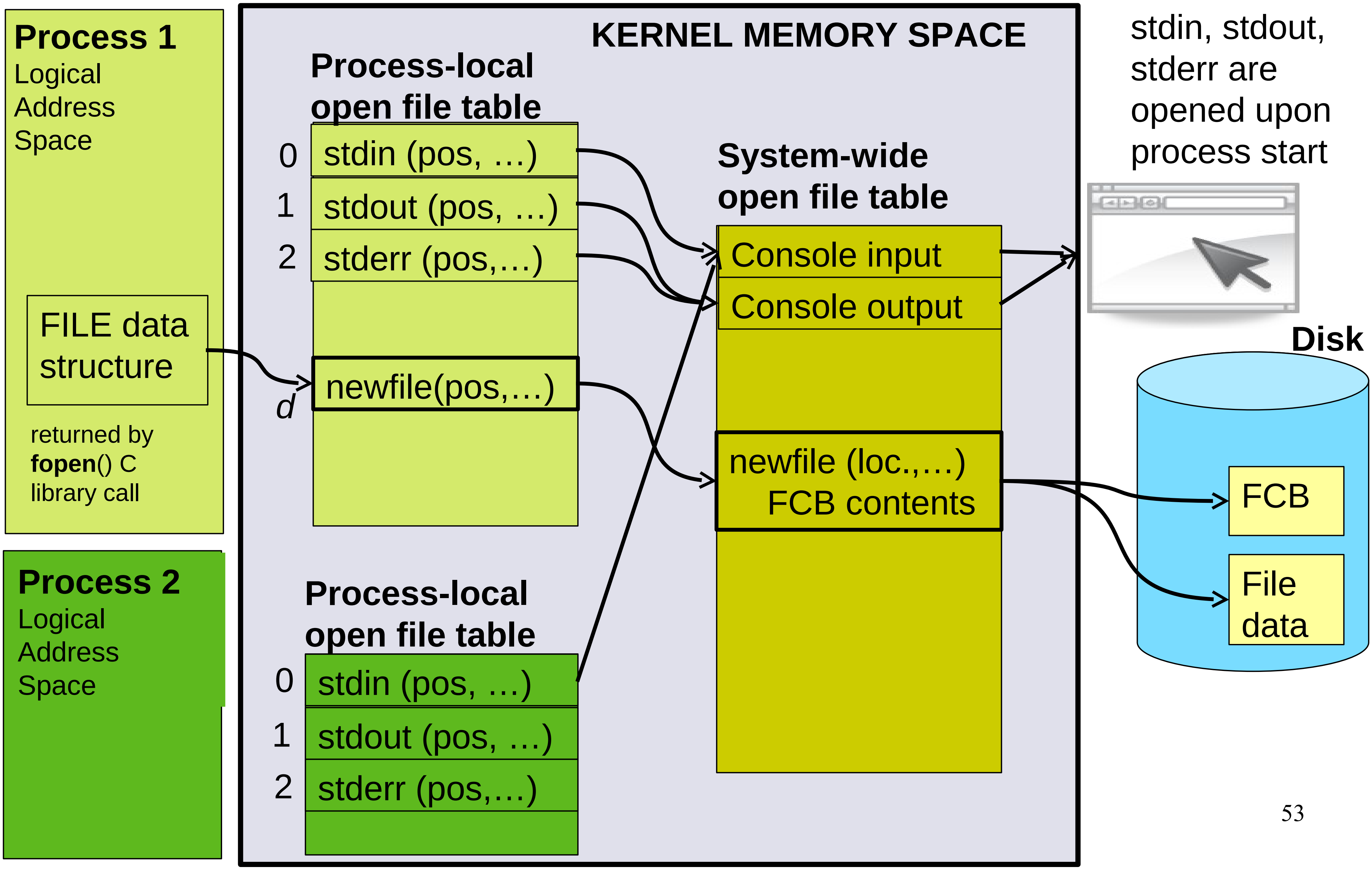
Open in Unix:

open (”filename”, ”mode”)

returns a ***file descriptor / handle*** = index into a per-process →
table of open files (part of PCB)

(or an error code)

File descriptors and open file tables



Storing open file data

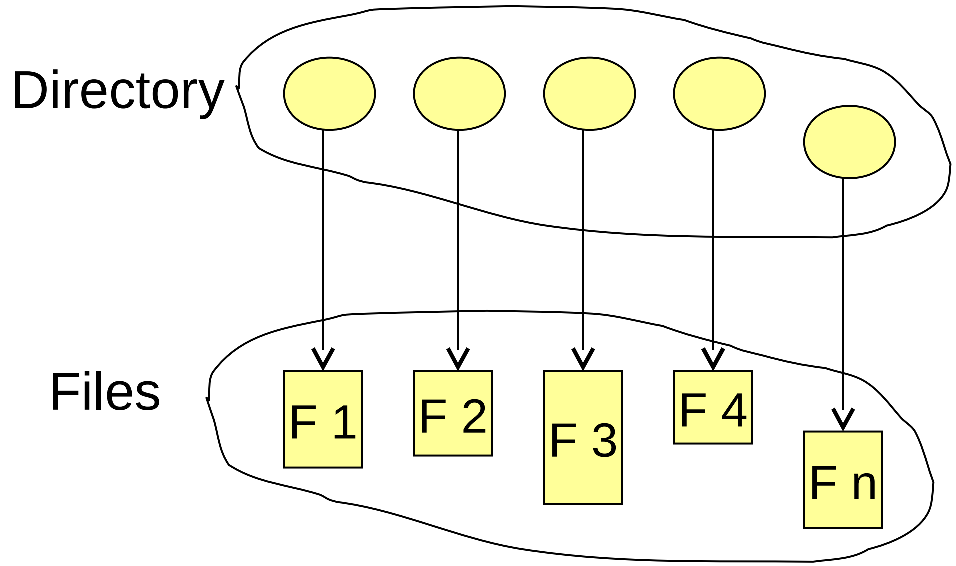
- Collected in a system-wide table of open files and process-local open file tables (part of PCB)

- Process-local open file table entries point to system-wide open file table entries

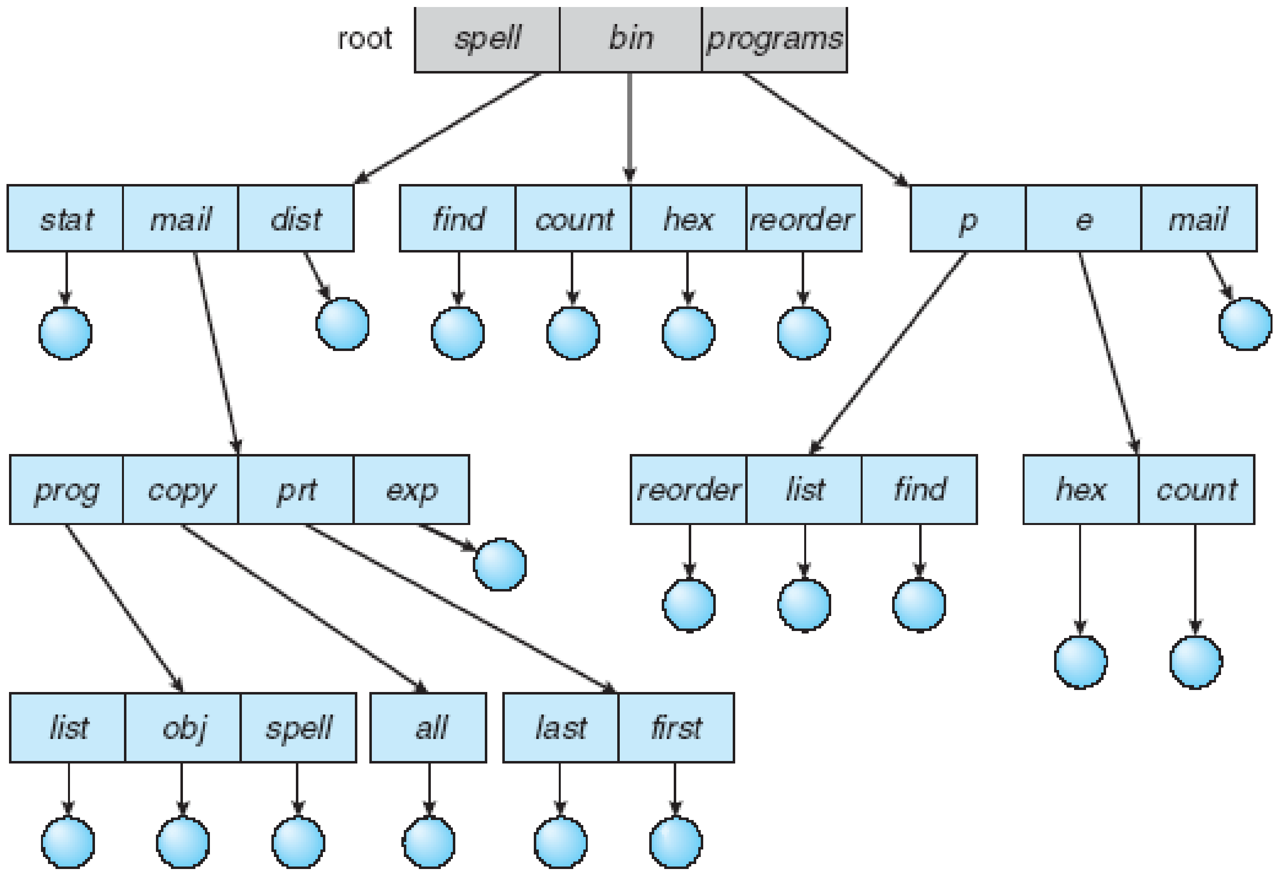
- Semantics of fork()?

Directory Structure

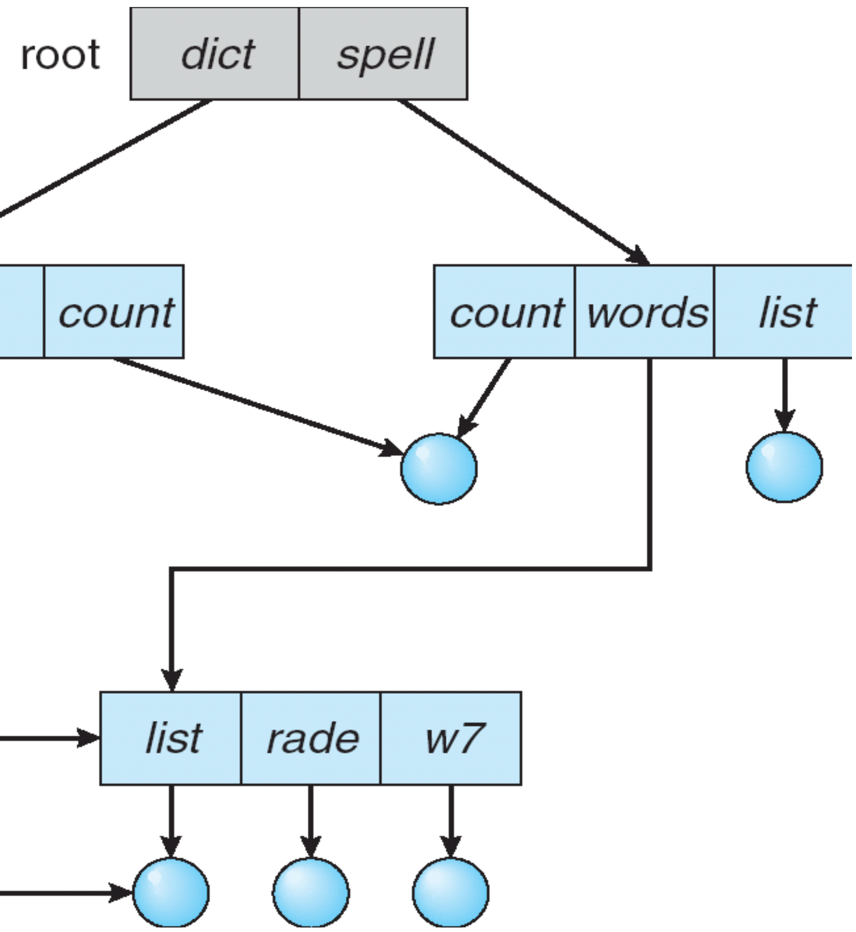
- Files in a system organised in **directories**
  - A collection of *nodes* containing information about all files
- Both the directory structure and the files reside on disk.



Tree-Structured Directories



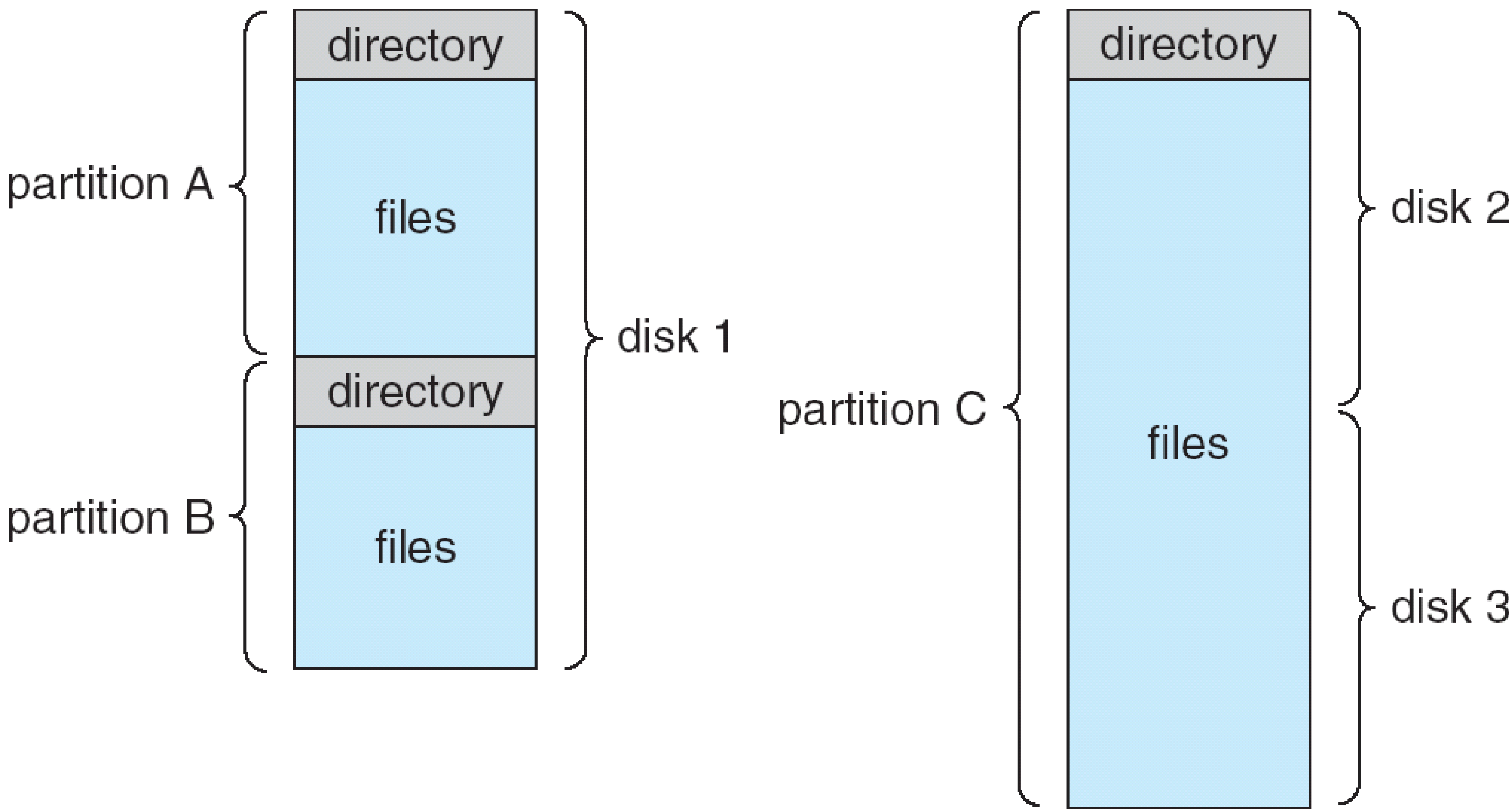
Acyclic-Graph Directories



Links

- **Hard links**
  - Direct pointer (block address) to a directory or file
  - Cannot span partition boundaries
  - Need be updated when file moves on disk
- **Soft links** (symbolic links, ”shortcut”, ”alias”)
  - files containing the actual (full) file name
  - still valid if file moves on disk
  - no longer valid if file name (or path) changes
  - not as efficient as hard links (one extra block read)

Examples of File-system Organization



File System Mounting

- A file system must be **mounted** before it can be accessed
- Mounting combines multiple file systems in one namespace
- An unmounted file system is mounted at a **mount point**
- In Windows, mount points are given names C:, D:, …

File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- In order to have a protection scheme, the system should have
  - **User IDs** - identify users,
    allowing permissions and protections to be per-user
  - **Group IDs** - allow users to be in groups,
    permitting group access rights

Sharing across a network

- Distributed system
- Network File System (NFS) is a common distributed file-sharing method
- SMB (Windows shares) is another
- Protection is a challenge!

What's next

- Next lecture: Lecture 4
  - Scheduling processes and tasks (good to know for Lab3 - suspended timer_sleep impl.)
  - Reading: Ch. 5.1-5.5, 5.8

- Lecture 5: Synchronization, tools to manage data sharing between multiple processes